Escuela Politécnica Superior

Gestión y sincronización segura de datos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Petar Alexandrov Nikolov

Tutor/es:
Rafael I. Álvarez Sánchez

Universitat d'Alacant
Universidad de Alicante

Junio 2018

Quisiera agradecer de antemano a las dos personas que han hecho este trabajo posible.

Por un lado, a mi pareja que me ha apoyado a diario y me he aguantado en momentos de estrés, y aguantado mis momentos de pensamiento y razonamiento sin sentido.

Y a mi tutor, que me ha ayudado en cada paso que daba a lo largo del proyecto, siempre atendiéndome y guiándome.

Índice

Índice de figuras

1. Introducción

Datos. Hoy en día los datos mueven el mundo. Los datos representan a las personas en el mundo cibernético. Cada persona genera cada día más y más datos que desea proteger. Fotos, vídeos, documentos, etc. Pero los datos se roban, se pierden, o son destruidos de forma accidental o intencionada.

Estos son los problemas a los que se enfrentan día a día los usuarios. Buscan almacenarlos en algún lugar seguro donde sólo ellos puedan tener acceso y no perderlos nunca, un lugar donde esta información esté realmente protegida. Los usuarios también desean, por otra parte, tener un acceso constante a esos datos desde cualquier tipo de dispositivo, y que todos ellos estén sincronizados.

Actualmente existen numerosos servicios en internet que combaten estos problemas ofreciendo sus productos. Pero algunos de estos productos, o más bien la mayoría, carecen de las características que un usuario debería de buscar en cualquiera de estos servicios: seguridad y privacidad.

Aunque todos los servicios proporcionen algún tipo de cifrado para los ficheros de un usuario, este tipo de cifrado no suele ser extremo a extremo, lo que significa que es la empresa detrás del servicio la que pone la clave de cifrado. Esto hace que sea seguro ante un atacante que intercepte la comunicación cliente-servidor, pero no para un atacante que consiga obtener la base de datos de claves del servidor, pues podrá descifrar todos los ficheros ahí almacenados; o para un atacante interno de la empresa. Tampoco otorga la privacidad y tranquilidad de saber que es el propio usuario, y dueño de los datos, el único que los puede descifrar y leer.

1.1 Justificación y objetivos

Como ya se ha descrito anteriormente, existe una problemática a la hora de almacenar todos estos ficheros, el hacerlo de forma totalmente segura, proporcionando la privacidad que un usuario espera y teniendo sincronizados todos los clientes de un mismo usuario.

Este será el problema que combatirá este trabajo, donde se explicará el proyecto desarrollado (en adelante, *kryle*), junto con todas sus características. Estas características, como se irán detallando a lo largo de este documento, estarán centradas en la protección de la información y en su fácil, rápida y transparente sincronización entre distintos clientes.

El objetivo principal de *kryle* será proporcionar al usuario una herramienta que pueda utilizar en su día a día, fácil de configurar, estable y segura para poder tener su propia *caja fuerte* de archivos donde sólo el usuario pueda tener acceso.

2. Estado del arte

A día de hoy, existen multitud de servicios para el almacenamiento de ficheros en la nube, es por ello por lo que, a continuación, se detallan algunos de los más prometedores y de los cuales toma ideas este proyecto (*kryle*). Estos servicios analizados son: NFS, Rsync, IPFS, Upspin.

2.1 Servicios de almacenamiento de ficheros

2.1.1 NFS

En primer lugar, analizaremos NFS (*Network File System*) (1). Es un sistema de ficheros desarrollado en *C* por *Sun MicroSystems* en 1984 cuyo objetivo era, y sigue siendo, acceder a ficheros remotos y tratarlos como si estuviesen en local. Actualmente está en su versión 4. Está diseñado para ser independiente de la arquitectura del ordenador, de la red, y del sistema operativo. Este servicio se basa en una arquitectura de comunicación de Cliente-Servidor.

Al ser un servicio desarrollado hace relativamente bastante tiempo, su modo de funcionamiento es bastante simple y es el siguiente:

- En primer lugar, el cliente ha de montar un volumen, recibiendo un descriptor con los nombres de ficheros, permisos, etc. Cuando obtiene este descriptor, el cliente ya puede trabajar a nivel local con estos archivos.
- Cuando una aplicación accede a algún fichero de los montados, envía la petición al VFS (Virtual File System) y éste manda la petición al módulo NFS, el cual hace una llamada a un procedimiento remoto (RPC) que envía al servidor, éste la procesa y edita (si tiene permisos de edición el cliente) el fichero en el disco y actualiza al cliente o clientes.

El hecho de que sea un servicio antiguo no lo hace idóneo para el uso diario de un usuario, ya que éste puede notar baja transparencia con el servidor, pues al abrir un archivo por primera vez, este comienza a descargarse y, en caso de ser un archivo muy pesado, este proceso no sería instantáneo y el usuario perdería tiempo.

No obstante, este sistema proporciona una gran ventaja frente a otras opciones, como el tratamiento de ficheros remotos en local, abstrayendo al usuario y a cualquier aplicación de la comunicación con el servidor, pues esto se hace a nivel de sistema operativo.

En cuanto a los casos de uso, NFS es utilizado muy a menudo para acceder a archivos comunes desde múltiples clientes, así como para acceder a archivos en equipos sin unidades de almacenamiento.

2.1.2 Rsync

Rsync (2) es una herramienta en *C* de copiado de ficheros muy rápida y versátil publicada en 1996 por Andrew Tridgell y Paul Mackerras, y mantenida actualmente por Wayne Davidson. Puede copiar ficheros tanto en local como en remoto. Es una herramienta famosa por su algoritmo *delta-transfer*, que reduce la cantidad de datos copiados pues solo transfiere las diferencias entre el fichero de origen y el de destino (sólo en redes donde la velocidad de transmisión es menor que el costo de realizar dicho algoritmo; en local, por ejemplo, puede ser más rápido copiar el fichero entero). Se usa como herramienta mejorada de copiado, para realizar *backups*, o para *mirroring*.

Rsync emplea dos modos de comunicación:

1. Mediante *pipes* (tuberías): ya sea en local, o bien un *pipe* entre una *remote shell* y el cliente.
2. Mediante sockets (cliente – sevidor(*daemon*))

De este sistema de ficheros nos centraremos en su algoritmo *delta-transfer* (3), el cuál puede ser muy interesante para nuestro proyecto de gestión de archivos. A continuación, se explica su funcionamiento:

Cuando un cliente B quiere actualizar los ficheros de un servidor A, lo primero que hace es calcular, para cada bloque del fichero a actualizar (entre 500 y 1000 bytes por bloque, dicho tamaño es ajustable), un *checksum* y un hash *MD5*. Estos son enviados al servidor.

El servidor, mediante el algoritmo *rolling checksum* (4), va comprobando cada parte del fichero (empezando en cada byte de forma secuencial) si coincide o no con el *checksum* del cliente. De ser así, calcula el hash *MD5* para cerciorarse de que efectivamente son iguales. Esto se debe a que el algoritmo del *rolling checksum* es débil y puede dar un falso positivo, es decir, que dos trozos distintos den un *checksum* igual. En caso de que sean distintos, el algoritmo del *rolling checksum* avanza un byte más. Esto se repite hasta que encuentre una coincidencia de *checksum* y hash *MD5*. En ese momento, envía al cliente B los bytes anteriores desde la última coincidencia encontrada (es decir, todos aquellos que difieran del cliente) junto con información de dónde debe el cliente indexar dichos "trozos" de bytes. Cuando encuentra una coincidencia, avanza hasta el último byte del "trozo" analizado, más 1.

La principal ventaja es la rapidez con la que transfiere los ficheros (o, más bien, los actualiza). Si la red de comunicaciones es demasiado rápida (por ejemplo, en local) transfiere el fichero entero, si no, transfiere sólo las partes que difieran de un fichero y otro. Esto permite poder actualizar ficheros con un menor ancho de banda.

Como principal desventaja, encontramos el cálculo computacional asociado al algoritmo *delta-transfer*, el cual puede ser muy grande si el fichero a actualizar es demasiado pesado.

2.1.3 IPFS

Seguidamente, analizaremos un sistema de ficheros más moderno y actual que pretende sustituir a la *Web* actual. Su nombre es IPFS (*InterPlanetary File System*) (5).

IPFS es un sistema de archivos P2P (*peer-to-peer*) desarrollado por *Protocol Labs* en *Go* y aún sigue en proceso de desarrollo. Cada nodo almacena objetos (representan ficheros y otras estructuras de datos). Si el objeto es menor a 1KB, se almacena dicho objeto; si no, se almacena la dirección física del nodo donde está el objeto. Para esto, IPFS se basa en una DSHT (*Distributed Sloppy Hash Table*).

La comunicación se basa en la conexión de nodos y la transferencia de estos objetos. IPFS no supone el uso sobre IP, sino que puede actuar sobre cualquier protocolo de red, pudiendo encapsular uno dentro de otro.

IPFS también posee su propio sistema de nombres para poder indexar y localizar los bloques. Este sistema de nombres se llama IPNS.

Su sistema de funcionamiento aún está en desarrollo pero, de momento, se basa en un sistema de crédito para promover el intercambio de objetos entre nodos. Si un nodo requiere de un fichero muy grande cuyos bloques están en distintos nodos, para que éstos los envíen, aunque no requieran nada a cambio, se diseñó IPFS con este sistema de crédito. Este sistema se basa en una relación entre cantidad de bytes recibidos y cantidad de bytes enviados. Si se reciben muchos, pero se envían pocos, la "deuda" de este nodo será alta y los otros nodos dejarán de enviarle bloques. La deuda de cada nodo se intercambia en el establecimiento de la conexión entre 2 nodos y, si esta difiere de la que tenía un nodo almacenado sobre el otro, el primero puede decidir si cerrar la conexión.

Actualmente, IPFS sigue en desarrollo, por lo que aún no se ha implementado la mejor estrategia para el intercambio de objetos entre nodos, no obstante, de momento se utiliza la *ratio de deuda* anteriormente explicado.

IPFS también permite almacenar en caché algunos bloques para agilizar el proceso de búsqueda, pues si a un nodo se le pide un bloque y no lo tiene, ha de buscarlo (con menos prioridad que sus propias búsquedas) para entregárselo al nodo solicitante de dicho bloque.

Como todo sistema descentralizado, la mayor ventaja que posee es la de no perder conectividad en caso de que un nodo falle, pues quedarán muchos otros que pueden seguir ofreciendo el servicio. Otra ventaja encontrada en este servicio es que, a diferencia de BitTorrent, un nodo puede pedir cualquier bloque de datos, independientemente del fichero al que corresponda dicho bloque.

Como desventaja, encontramos el sistema de autenticación, que consiste en el intercambio de un id y un hash (siendo éste el hash del id), y el otro nodo debe calcular el hash del id y comprobar que efectivamente son iguales. Este sistema no proporciona mucha seguridad y con relativa facilidad se puede suplantar a otro nodo.

Como ya se explicó anteriormente, un nodo puede almacenar un objeto (bloque de datos), o una dirección de dónde está dicho objeto. En caso de caer un nodo, el sistema seguiría funcionando, pero sus bloques no se podrían obtener hasta que el nodo se levante de nuevo.

2.1.4 Upspin

Upspin (6) es, o pretende ser (pues sigue en desarrollo), un sistema de archivos implementado en *Go* unificado para todo el mundo. Upspin no es un servicio, es un sistema de protocolos, una interfaz y una serie de componentes.

El objetivo de este sistema es tener un único sitio donde cada usuario pueda almacenar todos sus ficheros (esto puede ser en un servidor externo o en el propio

servidor/ordenador del usuario) y tener un control total sobre con quién o qué compartir dichos ficheros y con qué permisos hacerlo.

Upspin proporciona una interfaz que deberían de implementar los distintos servicios online (Dropbox, Mega, Google Drive, Facebook, etc.) para poder funcionar correctamente con él. Al hacer esto, el usuario puede subir las fotos de sus vacaciones (por ejemplo, en */fotos/vacaciones/2017*) y darle permisos de lectura a Facebook y de lectura y escritura a Google Drive, de esta forma el usuario tiene las fotos en un único lugar donde él tiene el control total de sus archivos.

Upspin cuenta con 3 componentes clave para su funcionamiento:

- *Key Server*: servidor donde se almacena la clave pública de cada usuario y la dirección del servidor de directorios.
- Servidor de directorios: servidor donde se almacena la dirección física del servidor de almacenamiento que contiene los ficheros a los que apunta el de directorios. También almacena metadatos como quién tiene acceso (autorización), y la clave de descifrado cifrada con la clave pública del propietario.
- Servidor de almacenamiento: servidor donde se almacenan los datos (ficheros). Upspin calcula que existiría 1 servidor por persona/familia/organización. Los ficheros se indexan por el hash de su contenido.

Con estos componentes:

1) Un usuario pregunta al *key server* dónde está el servidor de directorios para localizar el árbol de directorios del usuario.
2) Pregunta al servidor de directorios dónde está (físicamente) el fichero al que quiere acceder y éste le responde.

3) El usuario se conecta al servidor de almacenamiento para visualizar/descargar/borrar/editar el fichero.

Cada fichero está cifrado (7) con AES-256 en modo contador (CTR) en flujo y una clave aleatoria. Esta clave es cifrada con la clave pública del usuario propietario y almacenada en el servidor de directorios. El algoritmo usado para esto es la curva elíptica p256. Si un usuario desea compartir un directorio (la granularidad de Upspin es por directorios, no por ficheros) con otro usuario o servicio, añade su nombre junto con los permisos que tiene en un fichero *Access* y comparte su clave con dicho usuario o servicio. El mecanismo de compartición de clave consiste en descargarse la clave de cifrado del servidor de directorios, descifrarla con la clave privada del propietario, y cifrarla con la clave pública de la persona con la que se quiere compartir (obtenida del *Key Server*) y publicarla.

La mayor ventaja encontrada en Upspin es la comodidad y facilidad de gestión de contenido (8) que tiene un usuario sobre sus archivos. El usuario evita la duplicación de archivos en distintas redes sociales (por ejemplo, subir la misma foto a Facebook e Instagram). Del mismo modo, el darle permisos sobre algún directorio a otro usuario para compartir archivos o algún otro servicio es muy simple y rápido.

Los inconvenientes, por otro lado, son mayores. Upspin estima que se necesitará un servidor por usuario, lo cual es irrealizable por su gasto económico.

Además, Upspin permite otorgar permisos a directorios enteros, pero no a ficheros individuales. Continuando con el ejemplo anterior de las fotos de las vacaciones de un usuario, este puede haber tomado 200 fotos, pero querer subir solamente 20 a sus redes sociales. Para hacer esto, debe o bien duplicar esas 20 fotos en otro directorio y darle permisos de lectura a Facebook (por ejemplo) a ese directorio, lo cual entra en conflicto con el objetivo de Upspin (evitar la duplicación de datos entre redes sociales y/o dispositivos físicos o carpetas), o crear enlaces (ya sean simbólicos o fuertes) a los ficheros originales. Esta última opción es más

compleja para un usuario que no tenga un perfil técnico, por lo que se podría decir que Upspin está, de momento, orientado a usuarios con un conocimiento más avanzado.

2.1.5 Otros sistemas de ficheros

Como se ha podido ver en este análisis previo, existen multitud de servicios que ofrecen el poder almacenar los archivos en la nube con multitud de características diferentes. Existen muchos otros, como OneDrive, Dropbox, Google Drive, etc. Pero estos otros servicios se centran en lo mismo: conseguir que el usuario almacene sus ficheros en la nube y pueda editarlos online.

El hecho de que un usuario pueda editar un fichero online (es decir, sin necesidad de descargarlo, editarlo, y volverlo a subir) implica que ese fichero se encuentra en el servidor no cifrado (ya sea durante la edición o permanentemente) y, cuando está cifrado, lo está con una clave conocida por el servidor. Estas características implican una mayor comodidad para el usuario, pero una menor seguridad y privacidad al mismo tiempo. No obstante, otros servicios, como Mega o SpiderOak, sí que ofrecen un sistema de cifrado extremo a extremo.

Estos son los motivos por los que se han analizado los otros sistemas de ficheros, pues son los que nos proporcionaran las bases del diseño de nuestro proyecto, centrándonos en la seguridad y privacidad del usuario y de sus archivos.

3. Planificación del proyecto

En este apartado se detallará la metodología de trabajo, así como las herramientas necesarias para su desarrollo y prueba.

También se desarrollará una aproximación del coste de implementación, así como de la puesta en marcha y posibles líneas de negocio para su monetización.

3.1 Metodología de desarrollo

La metodología de desarrollo se ha basado en un modelo iterativo, donde tras la finalización de cada iteración se obtendría un resultado parcial del producto final, pero cada vez con más funcionalidades. Indistintamente de cuánto durase cada iteración, cada semana o cada dos semanas se ha quedado en tutoría presencial con el tutor para discutir las diferentes formas de abordar el problema en el que se estuviese trabajando en ese momento, así como discutir ideas sobre posibles mejoras futuras y para llevar un seguimiento del trabajo.

A lo largo del desarrollo de *kryle* surgieron problemas difícilmente solventables en el tiempo del que se disponía y se tuvo que hacer un rediseño parcial del núcleo del programa, por lo que la planificación final sería la siguiente:

Iteración 0 - Estudio de las tecnologías existentes y análisis de requisitos
0.1 Estudio del estado del arte y toma de ideas
0.2 Análisis de requerimientos
0.3 Definir alcance del proyecto
0.4 Estudio de las tecnologías para alcanzar estos objetivos
0.4.1 Estudio de los distintos modos de cifrado
0.4.2 Estudio de librerías para sincronizar archivos
0.4.3 Estudio de sistemas de comunicación seguros
0.5 Elaboración de un plan de desarrollo
Iteración 1 – Sistema de registro y login seguros
1.1 Estudio de los distintos modos de gestión de contraseñas

1.1.1 Estudio de funciones hash
1.1.2 Estudio de PBKDFs
1.2 Implementación de registro y login seguros
1.2.1 Uso de PBKDFs
1.2.2 Uso de un *nonce* aleatorio
1.2.3 Doble *hashing* en el cliente y servidor
1.3 Estudio de segundos sistemas de autenticación
1.3.1 Estudio de un *token* temporal enviado al correo
1.3.2 Estudio de certificados
1.3.3 Estudio de servicios terceros (*Latch*)
1.4 Implementación de un segundo factor de autenticación
1.5 Evaluar el correcto funcionamiento
Iteración 2 – Comunicación y almacenamiento
2.1 Análisis de los distintos protocolos de comunicación
2.2 Implementación de un sistema de comunicación fiable
2.2.1 Optimización para no cargar los ficheros en memoria antes de enviarlos
2.3 Estudio de los distintos modos de almacenamiento de ficheros en disco
2.4 Implementación de un sistema de almacenamiento de archivos en el servidor
2.5 Evaluar el correcto funcionamiento
Iteración 3– Creación de repositorios de archivos del usuario
3.1 Analizar modificaciones a incluir
3.2 Diseñar nueva solución
3.3 Implementar dicha solución
3.4 Evaluar el correcto funcionamiento
Iteración 4 – Cambio de diseño: carpeta central a sincronizar en el *home* del usuario
4.1 Analizar modificaciones a incluir
4.2 Diseñar nueva solución
4.3 Implementar dicha solución
4.4 Evaluar el correcto funcionamiento
Iteración 5 – Sincronizar automáticamente tras un cambio
5.1 Estudio de las tecnologías que permitan detectar cambios
5.1.2 Estudio de la librería *fsnotify*
5.2 Diseño de un protocolo que sincronice tras un cambio y al arranque
5.3 Implementación de dicho protocolo
5.4 Evaluar el correcto funcionamiento
Iteración 6 – Introducción de deduplicación a nivel de bloque
6.1 Estudio de las librerías que ofrece el lenguaje de programación para trabajar con ficheros
6.2 Estudio del mejor modo de fraccionar el fichero
6.3 Diseñar protocolo de deduplicación y sincronización de bloques
6.4 Rediseñar la base de datos del servidor

6.5 Implementar dichos diseños nuevos
6.6 Evaluar el correcto funcionamiento
Iteración 7 – Introducción de versionado
7.1 Diseño de un protocolo de actualización de índices
7.2 Implementación de interfaz para elegir versión a descargar
7.3 Implementación de protocolo de almacenamiento y tratamiento de versiones
7.4 Evaluar el correcto funcionamiento
Iteración 8 – Compartición de archivos con otros usuarios
8.1 Estudio de sistemas de compartición con clave pública
8.2 Implementación de sistema de compartición
8.3 Evaluar el correcto funcionamiento
Iteración 9 – Deduplicación a nivel de bloque con tamaño variable
9.1 Estudio e implementación del algoritmo *delta-transfer*
9.2 Implementación del algoritmo *Rolling-Checksum*
9.2.1 Evaluar el correcto funcionamiento
9.3 Implementación del algoritmo *delta-transfer*
9.3.1 Evaluar el correcto funcionamiento
Iteración 10 – Modificación del cliente como un *demonio*
10.1 Estudio de *demonios*
10.1.1 Cómo funciona
10.1.2 Cómo se implementa
10.1.3 Cómo se comunica el usuario con el *demonio*
10.2 Implementación del servicio como un *demonio*
10.3 Evaluar el correcto funcionamiento

Estas serían todas las iteraciones por las que habría que pasar para tener terminada la aplicación *kryle*. No obstante, y por falta de tiempo, nos quedaremos en la iteración 8, dejando las iteraciones 9 y 10 como ampliaciones futuras.

3.2 Entorno de trabajo, tecnologías a utilizar y herramientas

Este proyecto va a ser desarrollado, tanto la parte cliente como la parte servidor, en el lenguaje de programación *Go* (9).

Go es un lenguaje de programación desarrollado por *Google*. El código (no así el binario) de *Go* es multiplataforma, lo cual significa que para que nuestro cliente funcione en los distintos sistemas operativos solo lo tendremos que compilar con el compilador de dicho sistema. Es un lenguaje muy optimizado y fácil de utilizar, con un gran conjunto de librerías que facilitan mucho la programación. Es fuertemente tipado y compilado, hereda su sintaxis de *C*, es de código abierto y cuenta con recolector de basura. Su documentación es extensa y completa, con ejemplos de la mayoría de funciones.

Algunas de las librerías nativas del lenguaje que utilizaremos en este proyecto nos permiten cifrar ficheros, enviar correos electrónicos, detectar cambios en ficheros, realizar funciones *hash*, comunicación sobre *TLS*, codificaciones como base64, hexadecimal y JSON, etc., por lo que el uso de librerías externas se elimina prácticamente por completo.

El entorno de trabajo que utilizaremos será el editor de texto *Atom*, el cual, junto con algunos *plugins*, nos permite tener funcionalidades de *IDEs*, pero con la ligereza de un editor de texto.

En cuanto a las herramientas de seguimiento y control del trabajo, se han utilizado dos: *BitBucket* y *Trello*, ambas pertenecientes a la empresa australiana *Atlassian*. *BitBucket* es una herramienta que nos permite subir a la nube nuestros repositorios *git*, en este caso, hemos utilizado dos repositorios: uno para el servidor (*kryle-server*) y otro para el cliente (*kryle*).

El motivo de elección de *BitBucket* sobre *GitHub* es que nos permite tener tantos repositorios privados, de forma gratuita, como queramos.

Trello, por otro lado, es una herramienta más visual que nos permite tener varias listas de tareas personalizadas, *checklists* y demás. Nos permite clasificar las tareas como queramos, asignarles una descripción, una fecha límite de cumplimiento, etc. Esta herramienta ha sido de vital importancia la hora de evaluar en qué parte del proyecto nos encontramos, qué cosas tenemos hechas y cuáles no.

3.3 Modelos de negocio

En las siguientes líneas se analizarán algunos modelos de negocio que funcionan en otros servicios ya existentes y, una vez hecho el análisis, se propondrá el modelo de negocio que se seguiría con *kryle*.

- Anuncios: muchas empresas dejan que se anuncien terceros en sus aplicaciones para generar así ingresos. Esta es una técnica no válida para nuestro proyecto, pues es un proceso que se ejecuta como demonio y no posee interfaz gráfica en la cual mostrar dichos anuncios.
- Venta de información personal: como la mayoría de servicios online que no piden una cuota a sus usuarios (Facebook, Google, Twitter, LinkedIn, Instagram, etc.), se podría recopilar información del usuario y luego venderla a terceros. No obstante, esto entra en conflicto con uno de los objetivos de *kryle*, el cual es proporcionar un servicio que respete la privacidad de sus usuarios.
- Pagos por funcionalidades extra: si un usuario desea ampliar la funcionalidad de su cuenta, deberá de pagar una cantidad extra. Si es una funcionalidad concreta, como usar deduplicación para una sincronización más rápida, el usuario debería de pagar una única vez. Aunque esta funcionalidad requiera ejecutarse en el cliente, es el servidor el que decidirá (en base a si el usuario ha pagado por dicho servicio) si procesar los bloques que envía el usuario o procesar ficheros enteros. Esto es así debido a que el cliente será de código abierto y el usuario podría compilar el binario que incluya la funcionalidad de deduplicación. Por otro lado, si es una funcionalidad prolongada en el tiempo, se aplicará el modelo descrito a continuación.
- Pagos por aumento del espacio de almacenamiento por cuenta. Si el usuario desea tener un mayor almacenamiento en la nube, deberá de abonar una

cuota mensual/anual. Conforme el usuario requiera más espacio, esta cuota aumentará.

Tras analizar estos distintos modelos de negocio, optaríamos por el último de ellos, es decir, un modelo *Freemium*, donde cualquiera puede usar el servicio *kryle* pero, aquellos que requieran tener más espacio de almacenamiento, deberán de hacerlo suscribiéndose a una cuenta *Premium*.

Respecto a los pagos puntuales por funcionalidades extra, implantaríamos este modelo sólo para una funcionalidad en concreto, a la cual llamaríamos *Papelera de reciclaje*. Esta funcionalidad permitiría a un usuario poder recuperar archivos que haya borrado en los últimos 10 días. Esta funcionalidad sería de pago pues tendríamos que reservar en el servidor el doble de espacio contratado por cada usuario. No obstante, al suponer que dicha *papelera* no estará siempre llena, y se vaciará automáticamente cada 10 días, no lo contabilizaríamos como un servicio prolongado y por eso su pago sería único.

El motivo por el que no haríamos de pago la funcionalidad de deduplicación es porque sería uno de nuestros factores diferenciadores para que los usuarios elijan nuestro servicio y no el de alguna empresa competidora.

4. Desarrollo del proyecto

Una vez hecho todo el análisis previo, se dispuso al desarrollo propio de la aplicación. En los siguientes puntos se detallarán las funcionalidades del proyecto, junto con todos los diseños que se han hecho (almacenamiento general, deduplicación a nivel de bloque de longitud fija, versionado y deduplicación a nivel de bloque de longitud variable), un estudio de sus ventajas y desventajas, y una comparativa final entre todos estos sistemas de almacenamiento. De igual manera se expondrán las distintas medidas de seguridad que se han implementado para lograr un servicio seguro y que respete la privacidad del usuario.

4.1 Sistema de registro y autenticación

Como todo servicio que gestiona usuarios, estos han de poder crearse. Para esta funcionalidad, se ha estudiado cuál es la manera más segura de realizar la gestión de claves del usuario.

4.1.1 Autenticación

A la hora de autenticarse, existen diferentes formas de hacerlo, ya sea enviando la contraseña al servidor directamente, derivando de la contraseña del usuario una clave única y enviando esta en su lugar, certificados, etc. En nuestro proyecto, optaremos por el uso de una clave. Esta será derivada mediante funciones *hash* específicas (explicadas en el siguiente apartado) para así evitar tener que enviar la contraseña real por el medio de comunicaciones. Al hacer esto, evitamos que cualquier atacante pueda ver la contraseña elegida por el usuario para nuestro servicio (la cual puede haber utilizado en otros servicios, aunque esto sea muy poco recomendable) y vea solamente un *array de bytes* aleatorios carentes de significado.

A continuación, se muestra un esquema de este funcionamiento, señalando en rojo aquello que se evita y nunca ha de hacerse:

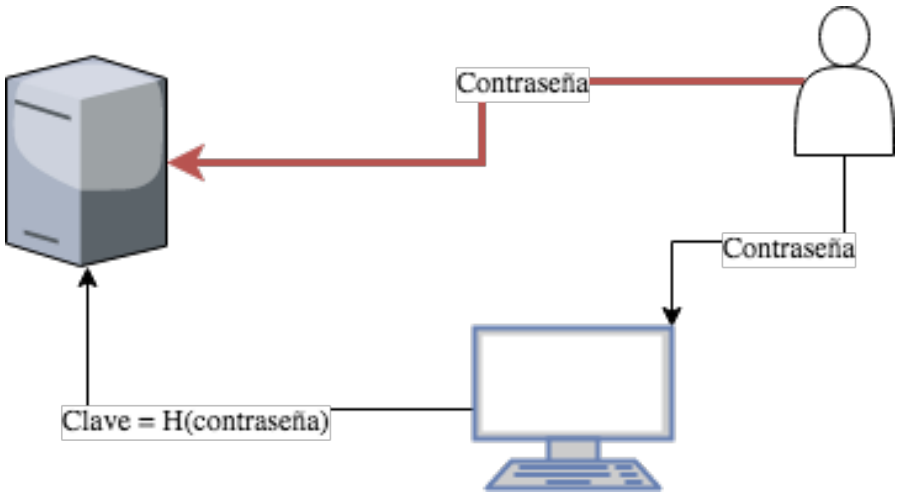

Figura 1- Autenticación del usuario

4.1.2 Almacenamiento de claves

A la hora de almacenar la clave del usuario en el servidor, tenemos varias opciones, estas son:

- En texto claro: aunque lo que se vaya a almacenar sea una clave derivada de la contraseña real, no debería de almacenarse en texto claro, pues como en los siguientes apartados explicaremos, las funciones *hash* tienen ciertas propiedades, entre ellas, producir la misma salida para la misma entrada, es decir, que tendríamos el mismo resumen almacenado para todos aquellos usuarios que hayan elegido la misma contraseña. Como se ha visto en casos anteriores (por ejemplo, el robo de la base de datos de claves de Adobe en 2013), con resúmenes iguales y los *hints* (pregunta predeterminada o frase escrita por el usuario para ayudar a recordar su contraseña en caso de olvido), se han podido recuperar las contraseñas utilizadas

por el usuario. Es por esto por lo que no se utilizará este método de guardado de claves en el servidor.

- Cifrada: un modo de almacenar la clave es cifrándola, pero para comprobar su veracidad cuando el usuario se autentique, deberá ser descifrada, lo que hace tener las mismas vulnerabilidades que el modo del apartado anterior. Además, al ser el servidor quien las cifre, si se roba la contraseña de cifrado, podrían descifrarse todas las contraseñas almacenadas, por lo que este método no nos aporta ningún tipo de seguridad.

- Función *hash*: una función *hash* es una función que, indistintamente del tamaño de entrada, siempre produce una salida de un tamaño fijo. Es fácilmente calculable y, dada la salida de la función *hash*, es imposible obtener la entrada, por lo que también se les llama "función de una dirección". El cliente calcula el resumen de la contraseña del usuario y es esta clave la que envía por la red para autenticarse. No obstante, hacer un *doble hash* en el servidor nos acarrearía los mismos problemas que si almacenásemos la clave resumida en texto claro, puesto que siempre nos daría la misma salida al resumir la misma clave y tendríamos los mismos problemas.

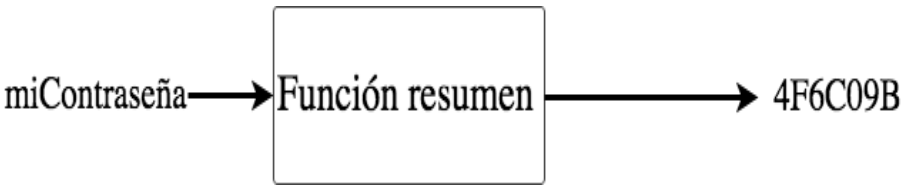

Figura 2 - Función hash

-Función *hash* + sal: puesto que la misma entrada produce la misma salida, un modo de solventar la vulnerabilidad de usar sólo el resultado de la función *hash* es añadirle una sal antes de calcular su resumen. Esta sal ha de ser aleatoria y distinta para cada usuario, de no serlo así y usar la misma para todos, nos llevaría al

problema anteriormente descrito. No obstante, el cliente cuando envía la clave al servidor, este ha de calcular el *hash* con la misma sal, sino produciría un resultado distinto. Es por esto por lo que la sal ha de guardarse también en la base de datos. La sal, de por sí, no otorga ningún conocimiento a un posible atacante ni permite obtener la contraseña original, por lo tanto, la sal puede ser guardada en texto claro en el servidor.

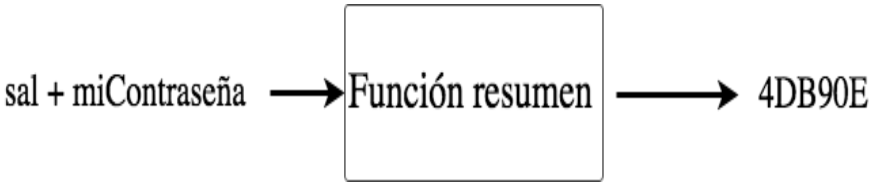

Figura 3 - Función hash + sal

- *PBKDF* (*Password-Based Key Derivation Function*): el método anterior es válido para evitar que un usuario malicioso intente averiguar cuál es la contraseña original, no obstante, como ya se ha mencionado antes, obtener el resultado de una función *hash* es computacionalmente fácil de obtener, lo que facilita el ataque por fuerza bruta. Para evitar este ataque, lo que se hace es ralentizar el cálculo del resumen de esta. El modo de hacerlo es utilizando un *PBKDF*, el cual es una función *hash* con algunas características que lo hacen más lento. Estas características son el consumo de memoria *RAM* (evitando así el ataque por *GPU*), la utilización de varios hilos o incluso puede tardar varios segundos (en la actualidad existen, por ejemplo, máquinas de minado de criptomonedas que pueden calcular 10 *TeraHashes* por segundo, lo cual facilitaría mucho el ataque si se utilizase una función *hash*, pero de poco serviría si se ralentizase tanto el ataque que en un segundo sólo se pudiese calcular un único *hash*).

- *PBKDF* + sal: al igual que ocurre con el ejemplo de utilizar sólo funciones *hash*, aquí ocurre también que para la misma entrada siempre produce la misma salida. De la misma manera, si le añadimos una sal antes de calcular su función

resumen, resolvemos la problemática con la que nos encontrábamos antes. Los últimos algoritmos creados de *PBKDF* requieren de forma obligatoria una sal, por lo que aumentan su robustez desde el diseño del propio algoritmo.

Hecho este análisis primero, hemos decidido utilizar para nuestro proyecto la última opción de almacenado de claves, es decir, *PBKDF* + sal.

El algoritmo utilizado para ello es *Argon2* en su versión *ID*, la cual combina el acceso a memoria independiente para la primera mitad de la primera iteración del algoritmo, y acceso a memoria dependiente para el resto. Esto hace al algoritmo resistente frente ataques de canal secundario (*side-channel attacks*: ataques basados en información obtenida del sistema operativo) y ataques de fuerza bruta. La utilización de este algoritmo en el lenguaje *Go* es la siguiente:

```
dk := argon2.IDKey(password, salt, 1, 64*1024, 4, 64)
```

Estos parámetros son:

- *password*: clave a la cual aplicar la función

- *salt*: sal que añadir a la clave antes de calcular su resumen

- *1*: tiempo que queremos que tarde la función en procesarse (en segundos)

- *64*1024*: cantidad de memoria a consumir (en KiloBytes)

- *4*: hilos a utilizar por el procesador

- *64*: tamaño de la salida producida (en bytes)

4.1.3 Cálculo en el cliente

Como ya hemos mencionado, la contraseña ha de enviarse resumida por el canal de comunicaciones. Análogamente a los casos anteriores, se puede enviar en claro, cifrada, resumida con una función *hash* o con un *PBKDF*, así como con una sal o sin ella. Puesto que la sal es aleatoria y el cliente no tiene dónde almacenarla, así como no proporcionar más seguridad pues solo se utilizará ese *hash* en la comunicación (el servidor volverá a hacer un *hash* de lo que le envíe el cliente), se podría utiliza una función *hash* o un *PBKDF* sin sal, en nuestro caso, usaremos una sal fija guardada en el código del cliente. No es tan importante ralentizar el ataque en el cliente como en el servidor, por lo que no pasaría nada al utilizar una función *hash*. No obstante, la clave maestra de cifrado del usuario será los 32 últimos bytes del resumen calculado (usaremos 32 bytes para AES-256), por lo que el resto sería la clave a enviar al servidor para autenticarnos.

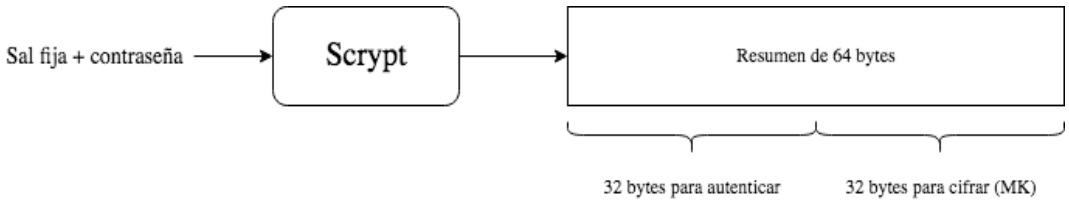

Figura 4 - Derivado de clave del usuario

El ataque por fuerza bruta consiste en probar todas las combinaciones posibles, y el *hash* más grande que se puede calcular es de 512 bits (64 bytes), por lo que para autenticarnos nos sobrarían 256 bits (32 bytes), lo cual significa que probar todas las combinaciones posibles es 2^{256}. Sin embargo, se están haciendo avances en los procesadores, tarjetas gráficas, e incluso en la computación cuántica, por lo que en un futuro tal vez no muy lejano esto sea un valor pequeño a romper por *clusters*. Además, se podría actualizar el estándar de cifrado en el futuro y requerir una clave de más de 32 bytes. Es por ello, que, para simplificar cualquier posible

cambio en el futuro, hemos optado por utilizar un *PBKDF*, concretamente, *Scrypt*, pues nos deja elegir el tamaño de salida (de momento lo hemos dejado en 64 bytes, igual que el *hash* más grande calculable).

4.1.4 Segundo factor de autenticación

Aunque utilicemos las funciones criptográficas más seguras para gestionar las claves de autenticado, esto sigue siendo poco, a día de hoy, para proporcionar un sistema seguro. Por ello se implantan otros factores de autenticación. Desde hace tiempo se dice: "una autenticación segura consta de algo que sabes (contraseña), algo que tienes (un teléfono o correo electrónico) y algo que eres (huella dactilar, iris del ojo, etc.)". En nuestro proyecto no hemos implantado el tercer factor de autenticación, pero sí el segundo. Este segundo factor de autenticación consiste en enviar un *token* (8 dígitos aleatorios) al correo electrónico del usuario con una vida de 3 minutos. Pasado este tiempo, el *token* caduca y otro será enviado en su lugar.

El funcionamiento de esto es sencillo: cuando un usuario inicia su autenticación y el servidor comprueba que la clave es correcta, envía un *token* al usuario. Si este lo introduce en menos de 3 minutos de forma correcta, entra a la aplicación y recibe una *cookie* de sesión para no tener que autenticarse en las próximas peticiones que haga; si falla, puede volver a intentarlo. En caso de que expire el *token*, uno nuevo será enviado en su lugar.

Este segundo factor de autenticación, aunque muy recomendable, es el usuario el que elige si activarlo o no. Puede activarlo tanto en el registro como más adelante.

En las siguientes capturas de pantalla se muestra cómo se registra un usuario, así como una autenticación válida y una no válida:

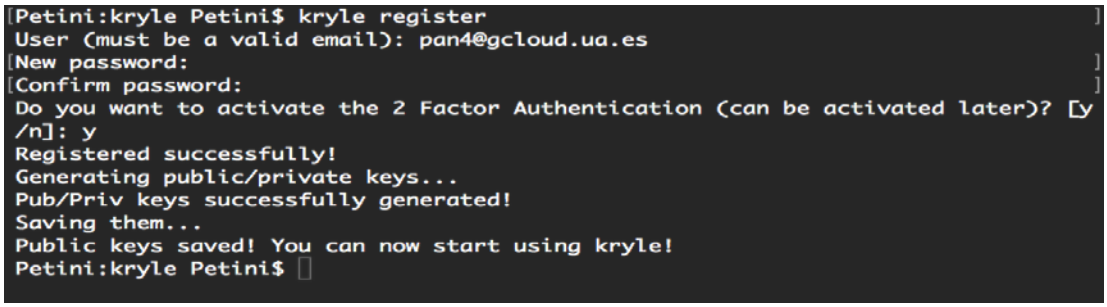

Figura 5 - Registro de usuario

Como se ve en la figura 4, hay una parte referente a claves públicas y privadas. Esta será explicada en el apartado 4.5 de este documento.

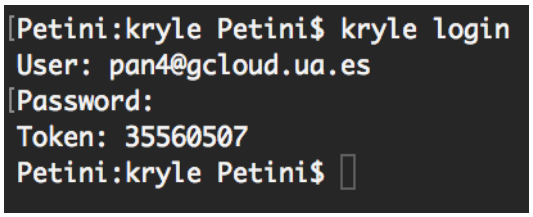

Figura 6 - Inicio sesión correcto

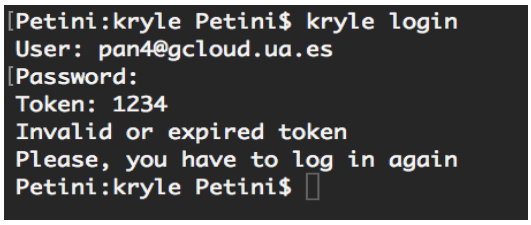

Figura 7 - Inicio sesión incorrecto

4.1.5 Resumen proceso de autenticado

A continuación, se muestra una imagen resumiendo cómo sería el proceso completo de autenticado de un usuario:

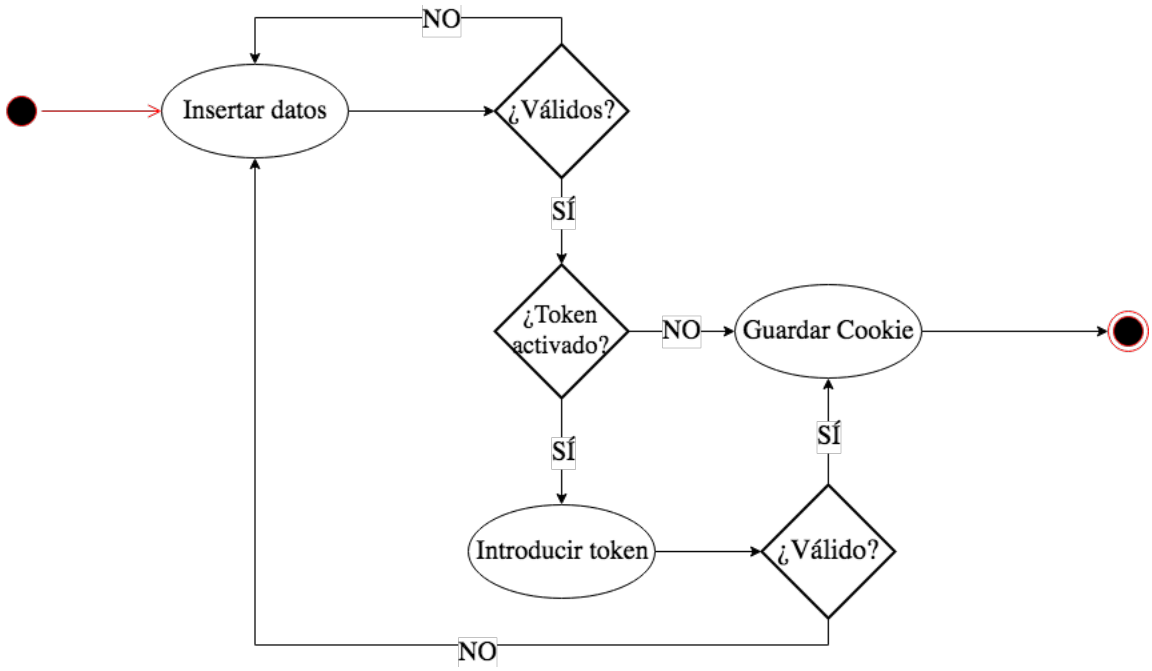

Figura 8 - Proceso de autenticado completo

4.2 Sincronización cliente – servidor

Una vez creado el sistema de registro y autenticado seguro de usuarios, pasamos a la implementación de un cliente que sea capaz de detectar cambios en los ficheros de manera automática (sin interacción del usuario) y subir al servidor los cambios introducidos, o bien descargar de él los cambios realizados desde otro cliente (por ejemplo, un segundo ordenador).

Para el desarrollo de este cliente se ha investigado en internet las posibles maneras de detectar cambios en un fichero. La primera aproximación era hacer un bucle infinito que comprobase la fecha de modificación de cada uno de los ficheros, comprobando así si ha sido actualizado por el usuario o no. Este primer algoritmo era ineficiente a la vez que poco escalable. Es por ello que, tras una investigación preliminar, se ha llegado a la conclusión de que la mejor manera era hacer uso de la librería *fsnotify* (10), muy utilizada en muchos lenguajes de programación para detectar cambios en ficheros sin tener que comprobar su contenido o fecha de modificación (*mtime*). Sin embargo, esta librería, para el lenguaje empleado (*go*) está en fase de desarrollo todavía y es inestable. No obstante, tras el desarrollo de este programa, se ha podido comprobar que funciona correctamente y no ha presentado ningún problema durante la implementación.

El funcionamiento de esta librería se basa en unos tipos de datos de *go* llamados *chan* (11). Un *chan* es un tipo de dato compuesto entre el tipo de dato original (*int, rune* (carácter Unicode), *struct*, etc.) y el propio *chan*. Este permite una comunicación entre varios hilos muy cómoda que facilita el desarrollo de software concurrente. Su funcionamiento es el siguiente (supongamos que hemos declarado una variable *chan* de tipo *int,* llamada *miInt*):

- *miInt* <-: cuando la "flecha" está en esta posición, significa que se le está asignando un valor a la variable. Un *chan* funciona como una cola, que puede almacenar muchos valores y, cuando se leen, se van leyendo desde el primero insertado hasta el último.

- <- *miInt*: cuando está así la "flecha", lee un valor dentro de *miInt*. Si no tiene ningún valor almacenado, se queda esperando de forma indefinida hasta que en otro hilo se le asigne un valor. Si no hay otro hilo asignándole valor, se produce un *deadlock* y el programa aborta.

Estos *chan* se combinan con una sentencia de control llamada *select*, la cual funciona como un *switch*: dados varios *chan* esperando a recibir valor (<- *miInt*), entra en el *case* del *chan* que primero reciba un valor. Por ejemplo, para manejar los posibles eventos sobre un fichero, este es el código empleado en el cliente:

```
done := make(chan bool)
go func() {
  for {
    select {
    case event := <-watcher.Events:
      log.Println("event:", event)
      switch event.Op.String() {
      case fsnotify.Chmod.String():
        updateFile(event.Name, client)
      case fsnotify.Write.String():
        updateFile(event.Name, client)
      case fsnotify.Create.String():
        addFile(event, watcher, client)
      case fsnotify.Remove.String():
        removeFile(event.Name, client)
      case fsnotify.Rename.String():
        if isDeleted(event.Name) {
          removeFile(event.Name, client)
        }
      }
    case errWat := <-watcher.Errors:
      log.Println("error:", errWat)
    }
  }
}()
addPathsToWatch(watcher)
<-done
```

Figura 9 - Escucha de eventos fsnotify

Si introducimos dentro de un *for* infinito un *select* que escuche varios *chan* (un *chan* de eventos y un *chan* de errores), cuando atienda un evento (por ejemplo, la creación de un nuevo fichero), volverá a la espera del siguiente evento (por ejemplo, editar un fichero). De esta manera, estaremos escuchando todos los eventos que se produzcan sobre las carpetas que se estén observando.

Cada una de las funciones mostradas en la imagen gestionan un tipo de evento distinto, ya sea subir un archivo, eliminarlo, etc. Estas funciones, internamente, hacen las llamadas pertinentes al servidor.

4.3 Deduplicación a nivel de bloque de tamaño fijo

Una vez realizado el cliente para detectar cambios de forma automática y que suba, descargue o elimine los ficheros sobre los que se efectuase algún evento, se continuó con el desarrollo hacia un diseño que trabajase con bloques de datos en lugar de con ficheros enteros.

4.3.1 Diseño de la base de datos

El primer paso para lograr este objetivo fue rediseñar la manera en la que se almacenaban los ficheros, así como las estructuras de datos. En la versión anterior se sincronizaba la carpeta del cliente con la carpeta en el servidor asociada para ese usuario, por lo que si el usuario *usuario1* creaba un archivo en */kryle/documentos/archivo.pdf*, en el servidor, en la carpeta de *usuario1*, se creaba el directorio */documentos* con dentro el archivo *archivo.pdf*. Esto, para la versión anterior, tenía sentido, no obstante, para esta, ya no. Se rediseñó la carpeta de cada usuario para que tuviese la siguiente estructura:

blocks/

files.json

index/

En la carpeta *blocks/* se guardan bloques de datos cifrados. Cuando el usuario crea un fichero nuevo (o edita uno existente), este cambio se detecta y se genera un evento, el cual es manejado por la aplicación cliente. Esta divide el fichero en bloques de 4MiB y los sube (el proceso completo será explicado más adelante) al servidor, con un nombre generado de forma puramente aleatoria con la función de *go* llamada *rand.Read* del paquete *crypto/rand*, la cual no debe confundirse con la función de idéntica signatura del paquete *math/rand*, pues esta última no está

considerada como segura para generar secuencias aleatorias. Esta secuencia generada, de 16 bytes, es codificada a hexadecimal {[0-9], [a-f]}, y el resultado es utilizado como nombre del bloque. Al principio del desarrollo de esta parte, era codificado a Base64 {[a-z], [A-Z], [0-9], +, /}, pero uno de sus caracteres (/) era interpretado como un separador de directorios por el sistema operativo y producía un error.

En el fichero *files.json* la información que se guarda, en formato *JSON*, es un *array* de *dedupl.File*. Esta estructura de datos es la siguiente:

```
//File struct with the data of a file
type File struct {
  Name   string    //file's name, e.g. document.pdf
  Hash   string    //file's content hash
  Mtime  time.Time //mod time
  Path   string    //path to the file in the client's app, e.g. /documents
  Length int64     //length, in bytes, of the entire file, Length%BlkSize to get
  //the last block length
  Index string //name of the index-S.json file
}
```

Figura 10 - Estructura de datos de un fichero

Es decir, por cada elemento del *array*, se guardan los datos del fichero: nombre, resumen del contenido, última fecha de modificación (se explicará más adelante el por qué de estos campos), ruta relativa de dónde se encuentra el fichero en el cliente, longitud del fichero en bytes y el nombre del índice que representa dicho fichero.

Por último, encontramos la carpeta */index*. En ella se guarda un índice por fichero. El nombre de cada índice viene dado de la siguiente manera:

$$HEX\left(E_{MK}^{AES}(path + file)\right)$$

Es decir, el nombre del índice del fichero X viene dado por la codificación en hexadecimal (no se usó otra codificación como Base64 por los motivos anteriormente descritos), del resultado de cifrar con AES (en los próximos apartados se explicará cómo se utiliza AES para cifrar) la ruta relativa del fichero y su nombre con la clave maestra del usuario, la cual se deriva a partir de la contraseña del usuario.

El contenido de este índice es, para cada fichero, un *array* de *string*, donde cada elemento del *array* es el resumen del bloque por el cual está formado dicho fichero. El orden en el que el cliente debe de juntar dichos bloques para formar el archivo original está determinado por el orden en el que aparecen los resúmenes de los bloques en el *array*.

Dentro de *index/* también hay un fichero llamado *b_index.json*, el cual contiene información sobre todos los bloques guardados en la base de datos. Esta información la almacena en un mapa (estructura clave -> valor), donde la clave es un *string* (que representa el resumen del bloque, es por esto que cada índice de un fichero está formado por un *array* de resúmenes, para poder indexar de forma rápida en este mapa y acceder a su clave de cifrado/descifrado o nombre con una complejidad constante y de forma sencilla) y el valor es una estructura de datos, la cual es la siguiente:

```
//Block each chunk of the file
type Block struct {
  Name string //name of the block
  Hash string //hex!
  Key  string //hex!
}
```

Figura 11 - Estructura de datos de un bloque

Esta estructura de datos está compuesta por el nombre que tiene el bloque en la carpeta *blocks/*, el resumen de su contenido, y la clave con la que está cifrado dicho bloque.

Los ficheros: *files.json*, *b_index.json* y cada índice que representa a un fichero están todos ellos cifrados con AES-256 empleando la misma clave (la clave maestra del usuario derivada a partir de su contraseña de autenticación). Y cada bloque dentro de *blocks/* está cifrada con una clave distinta, generada de forma puramente aleatoria, la cual está almacenada, para cada bloque, en el fichero *b_index.json*.

4.3.2 Políticas de actualización de ficheros

Una vez hecho el nuevo diseño de la base de datos, se ha procedido a la creación de un protocolo que decidiese qué ficheros actualizar cuándo dependiendo de las circunstancias, para así poder afrontar los posibles casos que se pueden producir.

En primer lugar, con ambos servicios iniciados (tanto servidor como cliente), es fácil deducir la política de actualización: si se detecta un cambio en un fichero, se procede a actualizar dicho fichero en el servidor (los algoritmos de actualización de ficheros están explicados en el siguiente apartado).

Otra política definida es, también con ambos servicios iniciados, descargar, cada cierto periodo de tiempo parametrizable (pueden ser 30 segundos, o 3 minutos, etc.) el fichero *files.json* del servidor y comprobar si el cliente está actualizado. Puede darse el caso de no estarlo si desde otra máquina el usuario ha añadido un fichero nuevo, por ejemplo.

Para definir las siguientes políticas de cómo actuar frente a archivos que ya no existen en el servidor (un archivo en el servidor no es más que un índice que referencia a un conjunto de bloques) o archivos que sí existen en el cliente, pero no en el servidor, se va a suponer que la versión del servidor **siempre** es la versión más actualizada. En el apartado de "futuras mejoras" se analizará como optimizar esto.

Si el usuario, con el servicio de sincronización apagado, edita un fichero, el *mtime* de dicho fichero se verá alterado y, cuando arranque el servicio, el algoritmo de actualización *onStart* actualizará dicho fichero en el servidor. No obstante, si el usuario añade un fichero con el servicio apagado, este no será subido al servidor y, al suponer que el servidor siempre está actualizado a la última versión, el algoritmo *onStart* será el encargado de borrar dicho fichero del equipo local cuando se arranque el servicio. De la misma manera, si un usuario desde un equipo borra un fichero sin estar conectado al servidor (ya sea porque no tiene acceso a internet, o el servidor está apagado por motivos de mantenimiento), dicho fichero no será borrado del servidor y, al conectarse al servicio, ese fichero sería automáticamente descargado del servidor.

4.3.3 Algoritmos de actualización

En esta sección se detallarán los algoritmos de cómo decide el cliente qué bloques subir y cuales descargar.

Empezaremos por el algoritmo ***onStart***, el cual se ejecuta nada más arrancar el servicio.

```
//always that we start the service, new Catalogue must be generated.
//maybe while it was shutted down, the user has changed something
localCatalogue := generateCatalogue()
srvFiles := parseFilesArray(client.GetKey(), client.Files())
updateSrvAndCli(client, localCatalogue, srvFiles)
```

Figura 12 - Algoritmo onStart

Como se puede ver en la captura, este algoritmo consta de 3 pasos:

1) generar un catálogo local con los datos de los ficheros: nombre, ruta, y *mtime*

2) obtener del servidor el fichero *files.json* y descifrarlo

3) comparar el catálogo de ficheros existentes locales con el almacenado en el servidor. En caso de encontrar un fichero más moderno en el cliente, lo actualiza en el servidor, y viceversa. En caso de tener un fichero en local que no está en el servidor, siguiendo las políticas de funcionamiento de *kryle*, se borrará. Y en caso de no estar el fichero en el cliente, pero sí en el servidor, se descargará.

A continuación, detallamos cómo funcionan las funciones de actualización de ficheros en el cliente o servidor.

Empezaremos describiendo el algoritmo de **subir** fichero.

1) El cliente descarga y descifra el índice de bloques del servidor. En él, recordamos, están guardados los nombres de los bloques, el resumen de su contenido y la clave de descifrado de dicho bloque.

2) De forma iterativa, el cliente lee el fichero bloque a bloque (de 4MiB de tamaño). Para cada bloque, calcula su resumen y comprueba, en la lista de bloques del servidor, si existe dicho bloque. En caso de existir, añade una referencia a ese bloque para el fichero. En caso contrario, cifra el bloque, lo sube al servidor, añade una referencia para el fichero y añade los datos del bloque al índice de bloques (*b_index.json*).
3) Una vez terminados de analizar todos los bloques resultantes de dividir el fichero, se sube al servidor el fichero *files.json* y *b_index.json* actualizados.

Este algoritmo es el mismo tanto para actualizar un fichero en el servidor que ha sido editado en el cliente, como para subir un fichero nuevo.

El siguiente algoritmo a describir es el de **descargar** fichero.

1) El cliente descarga del servidor la lista de bloques y la descifra.
2) Descarga, también, el índice que representa al fichero a descargar, y lo descifra.
3) Este índice, recordamos, contiene un *array* con el resúmen de cada bloque por el cual está formado el fichero. De forma iterativa, recorre dicho *array* e indexando en el mapa descargado de bloques (recordamos que el índice de bloques está formado por un mapa *resúmen -> struct*) obtiene el nombre del bloque que debe descargar, junto con la clave de descifrado de dicho bloque. Para cada iteración, descarga un bloque, el cual, tras descifrarlo, hace un *append* al fichero temporal que está formando en local. Para el último bloque, con el campo *Length*, realiza la operación *Length mod BlkSize* para obtener cuántos bytes útiles contiene este último bloque. Esto se debe a que todos los bloques ocupan 4 MiB, incluidos aquellos que contienen menos información. Una vez añadido este último bloque, el algoritmo termina y el fichero está correctamente formado en el disco duro del usuario.

Estos son los algoritmos utilizados para el proceso de subir/descargar archivos. Son invocados por la aplicación cliente dependiendo del evento que se dé.

El último algoritmo que queda por explicar es el algoritmo de **borrado** de ficheros. Cuando se genera un evento de este tipo, con ayuda del fichero *files.json*, se elimina el índice asociado a dicho fichero (recordemos que es en *files.json* donde está contenido el nombre del índice relacionado con qué fichero). No serán eliminados los bloques por los que está compuesto el fichero eliminado por dos motivos: que otro fichero puede estar haciendo uso de dichos bloques y que serán utilizados más adelante en el sistema de versionado, el cual explicaremos a continuación.

4.4 Versionado

Un sistema de versionado es un sistema en el que se guardan varias versiones de un mismo fichero. Por ejemplo, *Dropbox* es un ejemplo de servicio de almacenamiento de ficheros que proporciona versionado. En el desarrollo de software, un sistema muy empleado es *git*. Puede surgir la pregunta de por qué implementar un sistema de versionado en un servicio como *kryle* o *Dropbox*. En los siguientes apartados se dará respuesta a esta pregunta, que apunta a ser una necesidad en la actualidad.

4.4.1 Necesidad de un sistema de versionado

Al tener un conjunto de versiones por cada fichero que tengamos, nos garantizamos poder volver a un punto consistente anterior del fichero y poder retomar el trabajo desde ahí.

No obstante, este no es el principal motivo por el que se ha implementado esta funcionalidad. Desde hace 1 año, más o menos cuando surgió el sistema de pagos

anónimos que utiliza la tecnología *blockchain*, surgió un nuevo tipo de ataque. Este ataque es el llamado *ransomware*, que consiste en infectar una máquina y cifrar su disco duro y pedir un rescate económico para recuperar la clave de descifrado. Este rescate económico se pide a través de criptomonedas.

El tener una copia de seguridad en la nube en un sistema como *kryle* nos podría hacer pensar que, en caso de sufrir este ataque, nuestros datos podrían ser recuperados desde otra máquina. Sin embargo, *kryle* funciona con sincronización, lo que significa que, al realizarse un cambio sobre un fichero, este es subido automáticamente a la nube sustituyendo la versión anterior almacenada. Es decir, en caso de sufrir un ataque tipo *ransomware*, los ficheros del usuario serían cifrados y subidos al servidor de *kryle* cifrados, perdiéndose así la copia de seguridad.

Es imperativo, por este motivo, tener un sistema de versionado en un sistema con sincronización como *kryle*. De este modo, en caso de ser atacados y guardarse en el servidor una versión cifrada de los ficheros, desde otra máquina se podrían recuperar volviendo a la versión inmediatamente anterior a la versión corrupta.

4.4.2 Cómo funciona

Para implementar un sistema de versionado transparente al usuario, se ha tenido que modificar la estructura de la base de datos. Recordamos que su estructura era la siguiente (para cada usuario): una carpeta */blocks* con todos los bloques cifrados; una carpeta */index* con todos los índices de cada fichero, cifrados, y un fichero *b_index.json* con información referente a cada bloque (nombre, resumen, clave de cifrado); y un fichero *files.json* con información sobre qué ficheros existen, su longitud, resumen del contenido, etc.

La nueva estructura de la base de datos es igual, pero cambiando el contenido de los ficheros dentro del directorio */index*. Hasta ahora la información que contenía cada índice era un *array* de string (*[]string*) donde cada elemento era el resumen del bloque por el cual estaba formado. No obstante, ahora, este fichero contendrá información de cada versión, así como la versión actual.

```
//Versions struct, content of the index-X.json file
type Versions struct {
  Old     map[time.Time][]string
  Current []string
}
```

Figura 13 - Contenido de cada índice con versionado

Como se muestra en la imagen anterior, cada índice contendrá una estructura de datos con dos variables dentro. La primera de ellas, *Old*, hace referencia a todas las versiones anteriores de un fichero. Esta estructura es un mapa (clave -> valor), donde el valor por el que se indexa es la fecha de dicha versión, y el valor es, igual que antes, un *array* de *string* con el resumen de cada bloque por el que está formada dicha versión como elementos del *array*.

La otra variable, *Current*, hace referencia a la versión actual del fichero y es, como hasta ahora, un *array* de *string*.

Para generarse una nueva versión de un fichero, el usuario ha de modificar dicho fichero. Cuando lo hace, se genera un evento y, como hasta ahora, se suben los bloques nuevos a la base de datos. No obstante, ahora, la versión última del fichero (*Current*) pasa a indexarse dentro de la variable *Old* (como ya se vio anteriormente, en el fichero *files.json* se guarda el valor de *mtime*, el cual es utilizado como clave para indexar dicha versión en el mapa), y el nuevo *array* generado para la nueva versión se almacena en *Current*. Este nuevo índice es cifrado y guardado en la base de datos.

4.4.3 Cómo se usa en *kryle*

En este apartado se detallará cómo hacer uso del sistema de versionado en nuestra aplicación. Para ello, se ha creado una pequeña interfaz por línea de comandos para que el usuario pueda interactuar y elegir qué fichero con qué versión quiere descargar, y dónde hacerlo. A continuación, se muestra un ejemplo de uso de dicha interfaz:

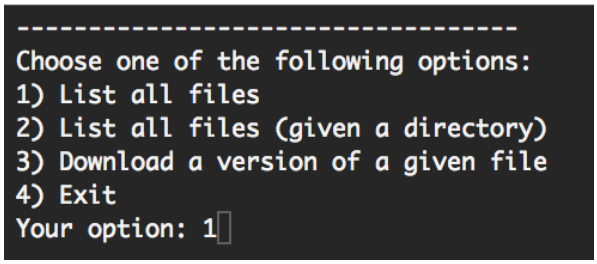

Figura 14 - Menú de versionado

Estas son las opciones que tiene un usuario cuando ejecuta el comando: *kryle vers*. En nuestro ejemplo, seleccionamos la primera opción de ellas, la cual nos imprime la siguiente salida:

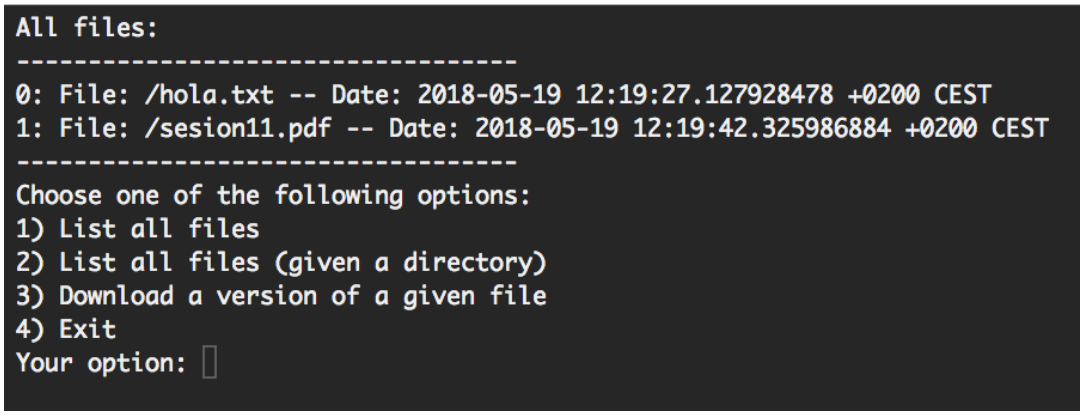

Figura 15 - Listado de ficheros

Una vez el usuario tiene información de qué ficheros tiene almacenados (junto con la fecha de la última versión, procede a seleccionar la opción 3, la cual la muestra el siguiente menú:

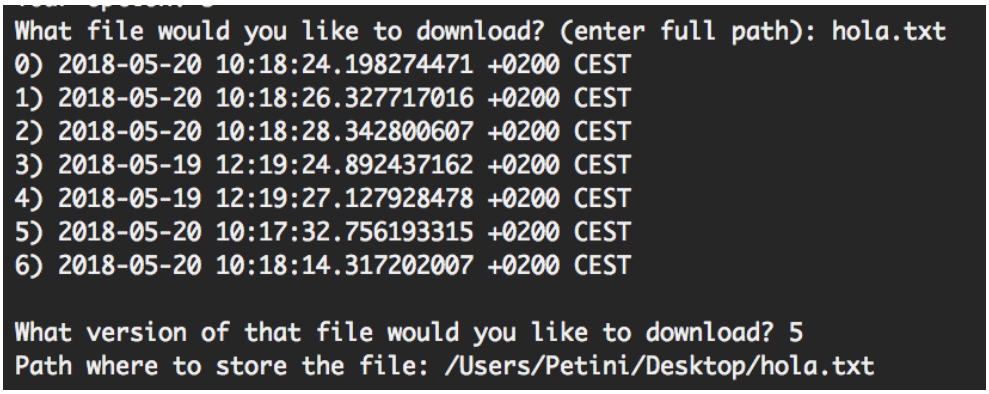

Figura 16 - Descarga de versión del fichero

En este menú se le pide al usuario que introduzca el nombre del fichero que quiere descargar y, a continuación, se le muestran todas las versiones almacenadas, junto con la fecha de cada una de ellas. Tras seleccionar una versión, se le pide que introduzca la ruta en la cual desea guardar dicho fichero.

Si el programa no imprime ningún mensaje de error, significa que todo ha ido bien y se le vuelve a mostrar el menú.

4.5 Compartición de ficheros

Otra funcionalidad implementada en nuestro servicio *kryle* es la posibilidad de compartir ficheros entre usuarios. Esta funcionalidad, tal cual se ha implementado, se podría identificar más bien como enviar ficheros; pues un usuario puede enviar a otro un fichero cualquiera pero, a diferencia de otros servicios como *Dropbox* o *Google Drive*, cuando se comparte un fichero entre dos o más usuarios, estos no tienen acceso al mismo fichero, sino que cada uno modifica su propia copia, por lo que el usuario que lo ha compartido no puede ver los cambios nuevos que efectúe el usuario segundo. En otras palabras, el usuario primero envía los bloques y claves de descifrado al usuario segundo y este lo guarda para él, sin sincronizarse los cambios del fichero entre dichos usuarios.

Para esto, como se mostró en la figura 5, cuando un usuario se registra, se crea un par de claves pública/privada y se almacena en su base de datos. Estas claves se almacenan en el directorio raíz de la base de datos con los nombres */pub.key* y */priv.key*. La clave pública, al ser pública, se almacena en texto claro, mientras que la clave privada, es cifrada con la clave maestra del usuario. A su vez, se crea una carpeta llamada */shared*. Es en esta carpeta en la cual se almacenarán los ficheros compartidos (o "regalados") por otros usuarios.

Para explicar el funcionamiento del sistema de compartición de archivos, lo plantearemos desde el punto el punto de vista de ambos usuarios (el que envía el fichero y el que lo recibe).

4.5.1 Enviando un fichero compartido

El procedimiento para compartir un fichero, al igual que en versionado, requiere de interacción por parte del usuario, por lo que se ha diseñado otro menú para que el usuario pueda elegir qué versión de qué fichero compartir con qué usuario.

En primer lugar, mostraremos el menú que tiene el usuario, al cual se accede ejecutando el comando *kryle share*:

```
Choose one of the following options:
1) List all files
2) List all files (given a directory)
3) Share file
4) Exit
Your option: 1
```

Figura 17 - Menú compartir fichero

Al igual que antes, el usuario, en primer lugar, ha de listar los ficheros que tiene para saber cuáles compartir:

```
All files:
----------------------------------
0: File: /hola.txt -- Date: 2018-05-20 10:18:29.122032522 +0200 CEST
1: File: /sesion11.pdf -- Date: 2018-05-20 10:18:14.566681848 +0200 CEST
----------------------------------
Choose one of the following options:
1) List all files
2) List all files (given a directory)
3) Share file
4) Exit
Your option: 3
```

Figura 18 - Ficheros listados en compartir

Una vez listados, procede a acceder al menú donde elige cuál enviar a quién:

```
What file would you like to share? (enter full path): hola.txt
0) 2018-05-20 10:17:32.756193315 +0200 CEST
1) 2018-05-20 10:18:14.317202007 +0200 CEST
2) 2018-05-20 10:18:24.198274471 +0200 CEST
3) 2018-05-20 10:18:26.327717016 +0200 CEST
4) 2018-05-20 10:18:28.342800607 +0200 CEST
5) 2018-05-19 12:19:24.892437162 +0200 CEST
6) 2018-05-19 12:19:27.127928478 +0200 CEST

What version of that file would you like to share? 4
Insert the user's name who will receive this file: otrocorreo@gmail.com
File succesfully shared
```

Figura 19 - Compartido fichero con éxito

Como se muestra en la figura 16, tras elegir qué versión de qué fichero compartir, se le pregunta al usuario con quién desea compartirlo, y este se envía.

Una vez entendido el proceso de cómo se comparte el fichero a nivel de usuario, explicaremos cómo se hace a nivel de protocolo:

1) En primer lugar, la aplicación cliente, tras obtener del usuario qué versión de qué fichero quiere compartir, se descarga del servidor el fichero *index/b_index.json* donde está la clave de descifrado de cada bloque. Esto es necesario puesto que al receptor del fichero compartido hay que enviarle los bloques, así como las claves de descifrado. Como ya se ha explicado, cada usuario tiene, por un lado, un índice que le indica qué bloques necesita para formar el fichero original y, por otro lado, contiene, en otro fichero, los datos de cada bloque (así evitamos repetir información en ficheros que comparten bloques). No obstante, al receptor del fichero debemos de enviarle toda esta información junta, por lo que los datos para formar el fichero se enviarán en el siguiente formato: *[]dedupl.Block*. Es decir, se enviará un *array* de bloques (ver figura 8) en el orden en el que debe de juntarlos el receptor.
2) Una vez hecho esto, se pide al servidor la clave pública del destino.
3) Después se genera una clave aleatoria de 32 bytes, la cuál será utilizada para cifrar con AES-256 el *array* explicado en el punto 1.
4) El resultado del punto 3 es enviado al servidor, indicándole en que usuario debe de copiar dicho *array* y qué nombre tiene el fichero original. De esta manera, el servidor, dentro de la carpeta *shared* del usuario receptor, crea una carpeta con el nombre del fichero. Es en esta carpeta donde guardará los bloques necesarios para formar dicho fichero, junto con un fichero, llamado *blocks.txt* que contendrá el *array* del punto 3.

5) A continuación, cifra la clave generada en el punto 3 con la clave pública del receptor y la envía al servidor para que la copie en el directorio adecuado (de nuevo, se le indica usuario receptor y fichero original para copiarlo en la carpeta correspondiente. Esta clave estará contenida en un fichero llamado *mk.key*.
6) Por último, el cliente genera un *[]string* (*array* de *string*). En este *array* se guarda, en cada posición, el nombre (no el resumen, como hasta ahora) de los bloques que conforman el fichero. Este *array* es enviado al servidor en texto claro (cifrado sólo por el canal de comunicación *TLS*) y es este el encargado de copiar los bloques de la carpeta *emisor/blocks/* necesarios a la carpeta *receptor/shared/fichero/*.

Con este algoritmo se evita que el servidor pueda conocer el contenido del fichero. El único dato que podría conocer sería el nombre original del fichero y de cuántos bloques de 4MiB está formado el fichero, pero no podría conocer el tamaño exacto pues todos los bloques ocupan lo mismo, incluso aquellos que sólo tienen 1 byte de información, por lo que sin conocer cuánto *padding* (relleno) tiene el último bloque, no podría conocer el tamaño real del archivo. Sin embargo, se plantea como mejora futura poder elegir otro nombre de fichero para confundir a un posible atacante.

4.5.2 Recibiendo un fichero compartido

En este apartado vamos a explicar el proceso de compartir fichero desde el punto de vista del receptor.

A diferencia del apartado anterior, donde el emisor tenía que interactuar con la aplicación, aquí el usuario receptor no ha de hacer nada para recibir los ficheros, de eso se encarga la aplicación cliente.

Por lo tanto, pasamos a explicar el protocolo de descarga de ficheros:

1) En primer lugar, el cliente hace una petición al servidor indicándole que le envíe una lista de cuántos ficheros compartidos tiene. Es decir, cuántas subcarpetas (con sus nombres) hay dentro de su directorio *shared*.
2) En caso de que haya algún fichero, procede con su descarga. Lo primero que hace el cliente una vez haya recibido la lista de ficheros es descargar su clave privada y descifrarla con su clave maestra.
3) A continuación, para cada archivo compartido: descarga el fichero *mk.key* y lo descifra con su clave privada. Luego, descarga el fichero *blocks.txt* y lo descifra con la clave guardada en *mk.key* y realiza un bucle. En este bucle, itera sobre el *array* que obtiene al descifrar *blocks.txt* y, para cada posición, descarga el bloque y, con la clave de dicho bloque, lo descifra y hace un *append* de dicho bloque al fichero que está creando en local. Cuando termina este bucle, envía una petición al servidor para eliminar dicha carpeta dentro de *shared*. Este nuevo fichero es almacenado en el directorio raíz del cliente y es trabajo de este de sincronizar el nuevo fichero con su repositorio de índices y bloques.

Este proceso de comprobar los nuevos ficheros compartidos se realiza una vez al arrancar el servicio (con el comando *kryle sync*), y una vez cada 60 segundos una vez iniciado el servicio.

4.6 Deduplicación a nivel de bloque de tamaño variable

Como una mejora que se podría implementar en un futuro, se plantea la posibilidad de deduplicación a nivel de bloque con un tamaño variable.

Esta mejora consistiría en implementar, como ya se describió en el estado del arte, subapartado 2.1.2, el algoritmo *delta-transfer*, pero sustituyendo la función resumen empleada por *Rsync* de *MD5* a otra más segura, como *SHA3-512*.

El implementar este algoritmo nos proporcionaría una sería de ventajas y desventajas que analizaremos a continuación.

En primer lugar, cabe destacar la gran ventaja que esto tiene: reducir el uso del ancho de banda del cliente. En nuestra implementación actual, si un usuario modifica un byte, es enviado un bloque de 4MiB al servidor (lo cual sigue siendo mejor que enviar el fichero entero). Si hiciésemos uso del algoritmo *delta-transfer*, aunque aumentaríamos el cálculo computacional requerido y disminuiríamos la velocidad de respuesta de nuestra aplicación, consumiríamos menos ancho de banda del cliente, lo cual haría que el uso de nuestro servicio *kryle* fuera más transparente para el usuario. Esto es así puesto que no notará una merma de calidad en su conexión a internet pues se enviarán menos datos por la red.

Otra ventaja encontrada es que, cuando un usuario edita un fichero, generaría bloques nuevos de distinto tamaño (posiblemente de tamaño más reducido), pero sin borrar los antiguos, pues son reutilizados para versionado. Al crear un fichero nuevo y tener que subirlo, este tiene más bloques con los que comprobar si los puede reutilizar o no. Esto es sobretodo útil para ficheros pequeños, donde tal vez ya haya un bloque de 500 bytes con el mismo contenido que el suyo. No obstante, el haber más bloques ralentiza la comprobación y aumenta el tamaño de *index/b_index.json*, por lo que tardaría más en transmitirse por la red este fichero.

Como gran desventaja encontramos el aumento considerable del tiempo de actualización (pues ha de ir comprobando trozo a trozo), así como el enorme cálculo computacional que se debería de hacer, sobretodo con ficheros grandes. Este cálculo sería más notable en equipos que no tienen un gran procesador o, por el contrario, en usuarios que utilizan nuestro servicio para trabajar y editar ficheros al mismo tiempo que estos son sincronizados. Usuarios que se dediquen a la edición de vídeo, por ejemplo, podrían notar una bajada del rendimiento de su máquina y de su productividad.

De igual manera, si un usuario actualiza de forma seguida y muy rápida alguno de sus archivos, la implementación actual permite terminar de fraccionarlo y actualizarlo antes de que empiece el manejador del segundo evento de "editado" a trabajar sobre el mismo fichero. Sim embargo, si hiciéramos uso del algoritmo de *rsync*, en ficheros grandes probablemente no daría tiempo a terminar de actualizar el fichero antes de que empiece el segundo evento, pudiendo así generar conflictos. Para evitar este problema, aumentaría el grado de complejidad de la implementación del cliente al tener que usar semáforos y bloquear algunos hilos hasta que el anterior termine.

Tras realizar este análisis, se podrían dejar implementados ambos sistemas de sincronización y dejar que sea el usuario, en tiempo de registro, el que decida qué sistema utilizar atendiendo a sus necesidades.

4.7 Elementos criptográficos

Durante el desarrollo de este documento se ha hecho alusión a AES-256, RSA y funciones resumen que no se han explicado. Es en este apartado donde se detallarán en qué modo se han utilizado estos algoritmos.

En primer lugar, explicar el algoritmo de criptografía simétrica que se ha utilizado. Este algoritmo es el estándar AES (el algoritmo se llama *Rijndael*) y se ha utilizado de la misma manera en los distintos puntos de la aplicación donde ha sido necesario. Esto es, utilizando una clave de 32 bytes, debido a que hemos utilizado AES-256, con el modo de operación CTR (modo contador), encadenando los bloques con la operación *XOR* para aumentar la entropía. Como vector de inicialización un *array* de 16 bytes generado de forma puramente aleatoria. Este vector ha sido almacenado al principio de cada fichero cifrado; así, al descifrarlo, se leía y se procedía al descifrado desde la posición 16 del fichero (el vector de inicialización ocupa las posiciones 0-15). Se ha especificado para cada apartado el modo de obtención de la clave de cifrado. En algunas situaciones como por ejemplo, cuando el cliente descarga el fichero *files.json* y lo vuelve a subir tras su lectura y obtención de información necesaria, o el fichero *index/b_index.json*, o cualquier otro fichero que sea descargado y posteriormente subido, a pesar de no ser modificado dicho fichero, se utiliza un vector de inicialización aleatorio cada vez que se cifran los datos, proporcionando una salida distinta a la que tenía anteriormente. De esta manera se mejora la seguridad en general.

En cuanto a la criptografía asimétrica empleada, el algoritmo utilizado ha sido RSA con una longitud de 3072 bits, lo que supone la longitud mínima recomendada, pues equivaldría en su nivel de seguridad a utilizar una clave de 128 bits en criptografía simétrica. En cuanto al modo de empleo de este algoritmo, se ha utilizado la versión 2 del modo PKCS, o comúnmente llamado, OAEP. Este modo

de funcionamiento comprueba, de forma automática, la integridad del mensaje descifrado.

Para ello, el desarrollador ha de especificar qué algoritmo utilizar para ello. En nuestro caso, hemos utilizado *SHA3-512*.

Por último, queda por mencionar que el algoritmo utilizado para calcular el resumen de los bloques ha sido *SHA3*, pues es el estándar que recomienda el *NIST*, con una salida de 512 bits.

5. Otras características implementadas

Hasta ahora se ha explicado el funcionamiento de *kryle*, así como de los algoritmos utilizados y los protocolos diseñados para su sincronización, versionado y compartición de ficheros. No obstante, hay varias funcionalidades implementadas que no han sido explicadas. Estas se detallarán en este apartado.

- Al igual que *git*, se pueden crear archivos de texto con *wildcards* dentro para excluir ficheros de directorios. Estos ficheros tienen el nombre de *.exclude.txt* y siguen la misma sintaxis que los ficheros *.gitignore*: un patrón por línea de los ficheros a excluir. Estos ficheros funcionan igual que en *git*: antes de sincronizar un fichero, se comprueba en su directorio si hay un fichero *.exclude.txt*. Si lo hay, se comprueba su contenido para ver si ese fichero debe ser ignorado o no. En caso de no haberlo, el cliente sube, de forma iterativa, directorio a directorio, buscando dicho fichero hasta llegar a la raíz */kryle*. Si no se encuentra ningún fichero (o se encuentra, pero el fichero no está excluido), se procede a su sincronización.
- Tanto en el cliente como en el servidor, existen ficheros de *log* que registran todas las acciones que ocurren en el sistema.
- El servidor cuenta con un fichero de configuración (el cual ha de indicarse su ruta al arrancarlo) mediante el cual se pueden parametrizar varios datos como: puerto de escucha, directorio raíz de la base de datos, ruta del fichero en formato *json* de la base de datos de usuarios, email (debe ser un g-mail) y contraseña para enviar el *token* al usuario y la ruta de dónde se encuentran los certificados para el servidor *TLS*.
- Al ser, tanto el cliente como el servidor en lenguaje *go*, automáticamente se utiliza la versión 2 del protocolo HTTP.

- Con motivo del nuevo Reglamento General de Protección de Datos (RGPD, por sus siglas en español), destacamos que nuestro servicio cumple con algunos puntos como, por ejemplo, no recopilar más información de la necesaria para que el servicio funcione. Esto es debido a que sólo pedimos al usuario su correo electrónico (necesario para identificarle, enviarle el *token* de inicio de sesión, y permitir el funcionamiento del sistema de compartición de ficheros) y una contraseña. Además, sería muy sencillo implementar un sistema que le permita al usuario descargar los datos almacenados sobre él en nuestros servidores (pues sólo sería responderle al usuario con su *email*). Todo esto es gracias a que nuestro servicio es de conocimiento cero y no puede acceder a ningún tipo de dato más allá del *email* del usuario.

6. Problemas encontrados

El desarrollo de *kryle* ha sido fluido y constante, sin estar muchos días sin trabajar en el proyecto ni atascarse en ningún aspecto. El desarrollo ha estado supervisado por el tutor y orientado por el mismo, no obstante, ha habido ciertos problemas durante el camino.

El primero de ellos, a nivel de librerías utilizadas, es el de la librería *fsnotify*. Esta librería es la encargada de capturar los eventos generados por el sistema operativo e informar de ellos a *kryle* para que pueda actuar en consecuencia. No obstante, esta librería aún está en desarrollo y tiene limitada la cantidad de directorios que puede "observar" y detectar eventos. Si se rebasase el número de directorios máximos permitidos, produciría un error y un mal funcionamiento del cliente.

El siguiente problema encontrado fue al principio del desarrollo. El diseño principal del cliente era parecido a *GitHub* o *Bitbucket*. Un servicio que gestiona varios repositorios (carpetas) que sincronizar con un servidor. Al sincronizarlos de forma automática (a diferencia de *git*, que sólo sincroniza por acción del usuario), complicaba mucho el diseño y el desarrollo de nuestro cliente. En algunos casos, durante su implementación, incluso generó conflictos con algunos ficheros. Este es el motivo por el que se cambió de un diseño por repositorios, a un diseño de una carpeta central (tipo *Dropbox*) que fuese esta la única a sincronizar (junto a sus subcarpetas).

7. Posibles mejoras

Al igual que todo producto *software*, siempre se puede mejorar y añadir nuevas funcionalidades.

Por un lado, tendríamos que mejorar la interfaz actual. Al disponer de relativamente poco tiempo para el desarrollo de este proyecto, se ha centrado la atención y el esfuerzo en conseguir que el servicio en sí funcione, atendiendo a temas de cifrado de datos, diseño e implementación de los diferentes protocolos (deduplicación, versionado y compartición, así como del autenticado del usuario), así como añadir varias funcionalidades explicadas en el apartado 5. No obstante, la interfaz del usuario se puede optimizar todavía, llegando incluso a implementar un cliente con interfaz gráfica. Este tipo de mejoras se han dejado para futuras ampliaciones del proyecto para poder centrarse en la problemática esencial de este trabajo.

Por otra parte, existen otras mejoras que se podrían introducir. Por ejemplo, si el servicio está encendido, pero no se tiene conexión a Internet, almacenar en una base de datos los eventos que se generan (junto la fecha de dichos eventos), para así, cuando el usuario vuelva a tener conectividad, poder descargar del servidor un registro de los eventos realizados (por lo que, también habría que llevar un registro de eventos individual para cada usuario) con su fecha, y comparar con la base de datos local para ver qué cambios se deberían de hacer. De este modo, se mejoraría la actual política de suponer que el servidor siempre es el más actualizado y descartar los cambios locales.

También se podría implementar la iteración 10 de la planificación, la cual, recordemos, consistía en convertir *kryle* en un *demonio*, facilitando así su operabilidad.

8. Conclusiones

Como se ha descrito en los apartados anteriores, se ha logrado diseñar e implementar un servicio de almacenamiento de ficheros que sincronice con el servidor y cualquier otro cliente del mismo usuario las modificaciones que éste realice. A su vez, estos ficheros son almacenados en bloques de un tamaño fijo (4MiB) para poder ser reutilizados por otros ficheros (deduplicación: eliminación de duplicados). Estos bloques no son eliminados, permitiendo así un sistema de versionado para que los usuarios puedan estar tranquilos frente ataques de tipo *ransomware*. Por último, el servicio *kryle* permite la compartición de ficheros con otros usuarios mediante el uso de criptografía de clave pública.

Todo esto se ha conseguido con un principio cada vez más necesario en el desarrollo de software actual: seguridad desde el diseño. La realización de este proyecto ha sido enfocada desde un punto de vista que respete la privacidad y seguridad del usuario, evitando así que el servidor conozca cualquier tipo de dato (pues toda la criptografía es siempre realizada en el cliente con claves que sólo él conoce).

Se plantean como mejoras futuras las ya explicadas en el apartado anterior, así como la implantación de un sistema de deduplicación a nivel de bloque variable.

Actualmente, el servicio más parecido al nuestro (en términos de seguridad, privacidad y funcionalidad) es SpiderOak (12), pues es de los pocos que proporcionan cifrado en el cliente. Por este motivo, se considera que nuestro servicio tendría una buena aceptación y utilización entre la comunidad de usuarios concienciados por la seguridad y privacidad de sus datos.

9. Referencias

1. **Institute for Advanced Professional Studies.** NFS. [En línea] http://www.iaps.com/NFSv4-new-features.html.

2. **Tridgell, Andrew.** Rsync. [En línea] https://rsync.samba.org/.

3. —. Rsync Delta-Transfer. [En línea] 09 de 11 de 1998. https://rsync.samba.org/tech_report/node4.html.

4. —. Rsync Rolling Checksum. [En línea] 09 de 11 de 1998. https://rsync.samba.org/tech_report/node3.html.

5. **Benet, Juan.** IPFS. *GitHub.* [En línea] https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf?raw=true.

6. **Upspin.** Upspin. [En línea] https://upspin.io.

7. —. Upspin Security. [En línea] https://upspin.io/doc/security.md.

8. —. Upspin Access Control. [En línea] https://upspin.io/doc/access_control.md.

9. **Alphabet Inc.** The Go Programming Language. [En línea] https://golang.org.

10. —. fsnotify. [En línea] https://github.com/fsnotify/fsnotify.

11. **Pike, Rob.** Concurrency Is Not Parallelism. *YouTube.* [En línea] 10 de 2013. https://www.youtube.com/watch?v=cN_DpYBzKso.

12. **SpiderOak.** Spider Oak. [En línea] https://spideroak.com/.

Glosario

Criptografía simétrica: hace referencia al sistema de cifrado donde la clave de cifrado y descifrado es la misma. Es un sistema de cifrado muy rápido con longitudes de clave (seguras) comprendidas entre 128 y 256 bits. El algoritmo más seguro, hasta la fecha, es AES (*Rijndael*). Su uso, por sus características, es el del cifrado de la información de propósito general.

Criptografía asimétrica: referente al sistema de cifrado donde la clave de cifrado y descifrado son distintas. Está compuesto por una clave pública (usada para cifrar) y una clave privada (usada para descifrar). Es un sistema muy lento de cifrado con longitudes de clave (seguras) a partir de 3072 bits. El algoritmo por excelencia utilizado es RSA. Su uso, por sus características, es el intercambio de claves y firma digital (este proceso es inverso al cifrado, pues se cifra con la clave privada para asegurar la autenticidad del emisor).

Deduplicación: sistema de almacenamiento de datos que consiste en no almacenar aquellos ficheros o bloques que estén repetidos.

Hash (resumen): función criptográfica de un solo sentido. Para la misma entrada siempre produce la misma salida con un tamaño fijo (depende del algoritmo a utilizar el tamaño de la salida). Es muy rápido obtener el resumen de cualquier fichero. Se usa para almacenar contraseñas (pues no se puede, dada la salida de esta función, obtener la entrada) y firmar digitalmente.

Conocimiento cero: sistema de almacenamiento de archivos (no confundir con el término *pruebas de conocimiento cero*) donde el servidor no conoce el contenido de los datos almacenados por el usuario, pues es este el que los cifra en el cliente.

Nonce: proviene del inglés *number used once* (número que sólo se utiliza una vez), también llamado *sal* o *vector de inicialización* (IV). Es un *array* de bytes que se

añade al principio de cualquier fichero (o datos) a cifrar para aumentar la entropía. Se usa, también, en algoritmos de resumen para evitar que dos entradas iguales produzcan la misma salida (a la hora de gestionar contraseñas). Es necesario almacenarlo para luego poder descifrar el contenido o volver a resumir una contraseña (por ejemplo) y obtener la misma salida. Sin embargo, el conocimiento de este *nonce* no aporta ningún conocimiento a un atacante, por lo que se puede almacenar en claro; lo importante, desde el punto de vista de la seguridad, es que no se repita.